INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An IP next generation compliant Java™ Virtual Machine

Guillaume Chelius, Eric Fleury

N°3936

May 22, 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3936--FR+ENG

An IP next generation compliant Java™ Virtual Machine

Guillaume Chelius*, Eric Fleury†

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n ° 3936 — May 22, 2000 — 34 pages

Abstract: IPv6, the *Internet Protocol version 6*, was decided to be designed, ten years ago, in order to replace the current version. It has now been almost completely specified and several implementations have been written for different platforms. However it is not widely used so far. One reason for this is the unavailability of softwares that can deal with IPv6. Thus the large use of the new protocol will only happen when enough attractive applications will be available.

In this report, we present two Java tools that are designed to ease the development of IPv6-able softwares. We concentrated our work on Java-written software because the use of this language is growing in a lot of network-based domains: distributed systems, parallel computing, high-performance computing, network management, active networks...

The first tool is a Java package that provides all the functionalities required to deal with IPv6. It has been designed with two main characteristics in mind. The first is the necessity for the underlying mechanisms to be very similar to the original ones. The second is to enable an easy transition between the code of an IPv4 software and its IPv6 counterpart. The second tool is an IPv6-able JVM, *Java™ Virtual Machine*. It gives every Java program the ability to deal simultaneously with IPv6 and IPv4, without any code modification.

Key-words: Network Programming, IPv6, IPv4 compatibility, Java, Internet Standardization

(Résumé : tsvp)

* gchelius@ens-lyon.fr
† Eric.Fleury@loria.fr

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ
Téléphone : 03 87 20 35 00 - Internatională: +33 3 87 20 35 00
Télécopie : 03 87 76 39 77 - International : +33 3 87 76 39 77

Une machine virtuelle Java™ compatible IPv6

Résumé : Il y a dix ans, la décision fut prise de développer IPv6, *Internet Protocol version 6*, afin de remplacer la version actuellement déployée. Ses spécifications sont maintenant quasiment complètes et plusieurs implantations sont supportées sur différentes plate-formes. Cependant il demeure très peu utilisé. Le fait est que très peu de logiciels supportent ce nouveau protocol et son utilisation ne se répandra pas tant que d'avantages de ressources ne seront pas disponibles.

Dans ce rapport, nous présentons deux outils Java qui sont supposés faciliter le développement de logiciels supportant IPv6. Nous avons concentré notre travail autour de Java parce que son utilisation est de plus en plus répandue dans des domaines aussi divers que les systèmes distribués, le calcul parallèle, le calcul de haute performance, la gestion de réseaux et les réseaux actifs...

Le premier outil est un package Java fournissant les mécanismes nécessaires pour supporter IPv6. Il a été écrit avec les deux principales caractéristiques suivantes. La première est le maitient d'une grande similarité entre les mécanismes internes à ce package et les originaux. La seconde est le support de transition facile entre le code d'un programme IPv4 et celui d'un programme IPv6. Le second outil est une JVM, *Java*™ *Virtual Machine*, supportant IPv6. Elle offre à chaque programme Java la possibilité de supporter à la fois IPv6 et IPv4 sans aucune modification.

Mots-clé : programmation réseau, IPv6, compatibilité IPv4, Java

1 Introduction

IPv6, for *Internet Protocol version 6*, is the next generation Internet network layer IP protocol often referred to as *IP next generation (IPng)*. IPv6 was designed to replace the actual IP protocol (IPv4). The decision for its development was taken nearly ten years ago and several research teams began to design it. At the same time, development of IPv6 stacks started.

Today, its evolution has reached a point that makes possible the work under IPv6. The existing stacks support enough platforms and offer enough functionalities to develop IPv6 networks. However, the transition from IPv4 to IPv6 has not really begun yet. Except the 6bone[1] which is a world wide IPv6 test-bed to assist in the evolution and deployment of IPv6, and few other exceptions, IPv6 is not widely implemented and fully tested for inter-operability.

One of the problems is that for IPv6, the number of available softwares is not satisfactory. Lots of softwares working under IPv4 do not have their counterpart under IPv6. This statement was the one which motivated us for finding a solution to the translation of IPv4 softwares to IPv6 and for easing the writing of new ones.

This goal can be obtain by two complementary approaches: providing a `IPv6.java.net` package (API for IPv6) and/or running a JVM above an IPv6 stack and thus inheriting all the enhancements of IPv6. We decided to write a Java package, `fr.loria.resedas.net6`, and a library `libjavanetipv6`, which, added to a *Java*^{TM} *Virtual Machine*, would provide to programs the possibility to deal with IPv6 as well as all new functionalities introduced in the Next Generation Internet Protocol. We concentrated our work on Java-written software because the use of this language is growing in a lot of network-based domains: distributed systems, parallel computing, high-performance computing, network management, active networks. Moreover, Java has potential to be a better environment for a lot of application development than any previous languages. One of the main reasons of the popularity of the Java programming language is its support for distributed computing [3]. Java's API for sockets, URL and other networking facilities is much simpler than what is offered by other programming languages like C or C++. Moreover, the Java^{TM} Remote Method Invocation (RMI) was designed to make distributed application programming as simple as possible, by hiding the remoteness of distributed objects as much as possible. All these APIs that were developed and used with an IPv4 JVM will be fully available on our IPv6 Compliant *Java*^{TM} *Virtual Machine*.

Before presenting into details the package and its underlying library, we give in section 2 a quick overview of the main characteristics of IPv6 and the new functionalities introduced in the Next Generation Internet Protocol. Section 3 will present our IPv6 java.net package. Section 4 details the native interface developed below the Java API. Section 5 is concerned with the results of this implementation. Section 6 gives some last updates and the availibility of the package. Finally, section 7 summarizes the achieved work.

[1] `http://www.6bone.net/`

2 A quick overview of IPv6

In the early nineties, in this context of technological improvement in both computer science and telecommunication areas, the expansion of the Internet has become such huge that the format of addresses was not adapted to the number of potential hosts any more. This was the main reason which pushed researchers to develop a new version of the IP protocol. In addition, this development was also a good way to introduce advances made in network research during the twenty five last years. These advances concern a wide domain covering security, flow control, mobility and multicast. All this work led to the conception of Internet protocol version 6 (IPv6 for short), also known as *IP next generation*. Much of the current IPv6 architecture has already been ratified, and implementations are emerging [9] even if – as an evolving technology – IPv6 is far from being a production network proposition at present. Several issues still remain.

In this section, the main characteristics of IPv6 are presented: the new addressing format and changes in the previous protocols, such as TCP, UDP or ICMP. Then new services will be presented with particular attention to IPsec, the protocol which provides security under IPv6. The goal of this section is not to list exhaustively all the details of the Next Generation Internet Protocol. For a full description of IPv6, please refer to the RFCs[2] or to [6, 18].

2.1 Addressing format

One can sum up the IPv6 challenge by "*Change while staying the same*". Developed over many years of careful design and exhaustive review, the IPv6 addressing scheme is radically new, based on the demographic nature of the community it will serve. At the same time, it includes provisions for upward compatibility from and inter-operability with today's IPv4 network architecture.

2.1.1 Address size and notation

The ability to sustain continuous and uninterrupted growth of the Internet could be viewed as the major driving factor behind IPv6. IP address space depletion and the Internet routing system overload are some of the major obstacles that could preclude the growth of the Internet. Even though the current 32 bit IPv4 address structure can enumerate over 4 billion hosts on (2^{32} = 4818200576) as many as 16.7 million networks, the actual address assignment efficiency is far less than that, even on a theoretical basis [13]. This inefficiency is exacerbated by the granularity of assignments using Class A, B and C addresses. By extending the size of the address field in the network layer header from 32 to 128 bits, IPv6 raised this theoretical limit to 2^{128} nodes (more than 1.000 addresses per square meter of the earth, ocean included). Therefore, IPv6 could solve the IP address space depletion problem for the foreseeable future. In practice, however, the size of the address is less important than its structure. IPv6 recognizes three kinds of unicast address, offers a new multicast address format, and introduces anycast addresses.

[2] `http://www.6bone.org/rfcs.htm`

Just as for IPv4 addresses, a more readable form for IPv6 addresses has been defined. Starting from the 128-bit format, one have to cut it in 8 words of 16 bytes, and give the hexadecimal representation of each one separated by semi-columns to obtain the new representation. For example, `3ffe:2c0:20:1:0:0:4796:23` is a valid IPv6 address. We will see later how to interpret it in details. To simplify the reading and writing of addresses, a condensed representation has also been specified, *i.e.*, when several zeroes are present, they can be compacted to `::`. Of course, this notation can be present only once in an IPv6 address.

2.1.2 Addressing plan

After finding the new storage length for addresses, the question was to choose an addressing plan. It had to provide a better structure to the Internet in order to avoid problems which had been encountered under IPv4; for example, the anarchic growth of routing tables.

The response given with IPv6 is a structure of the Internet in several hierarchic levels. To each level, different routing and addressing policies are associated, giving more flexibility in the addressing plan. Moreover each level should be aware of its sub-level in order to manage a better routing and to give its super-level the best understanding of the architecture.

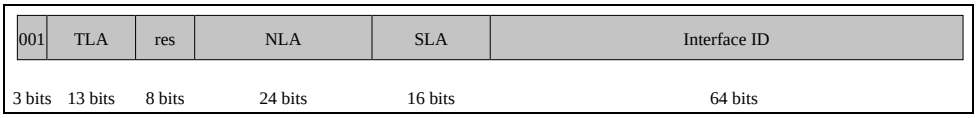

Figure 1: IPv6 unicast address format

After several propositions, like the *geographic-based unicast address* or the *provider-based unicast address*, the addressing plan that seemed to correctly reflect this architecture was found: the *aggregatable global unicast address format*. It is divided into three hierarchic levels: a public topology, a site level topology and an interface ID.

The public topology represents the IP connectivity providers described by the 48 first bits of the unicast address. The aim of the TLA (Top Level Aggregator) and NLA (Next Level Aggregator) fields that can be seen in figure 1 is to introduce several hierarchic levels in the public topology. The site level topology, corresponds to the local network. It is referenced by the SLA (Site Level Aggregator) field. Finally, the interface ID is used to identify different interfaces on a same local network. Its uniqueness is necessary only on a link. It is created using the IEE EUI-64 ID or the MAC IEEE 802 ID and a simple transform.

2.1.3 Other IPv6 addresses

In addition to the unicast address, there exist particular addresses in IPv6: for example the *unspecified address* which is used by a network node during its initialization, the *loopback address* which corresponds to the address 127.0.0.1 under IPv4, the *local-link address* which is only valid on a link, the *anycast addresses* which specifies a group of interfaces, the

IPv4-mapped address or the *multicast addresses*. Only *IPv4-mapped address* and *multicast addresses* will be described here.

IPv4-mapped addresses. This type of addresses, illustrated on figure 2, are constructed from classical unicast IPv4 addresses. Their use is to provide for an IPv6 stack a vision of the IPv4 world. With a dual stack, whether concept is discussed in section 6.1, an IPv6 application can communicate under the IPv4 protocol with other computers just by handling *IPv4-mapped addresses*. It becomes possible to write protocol-independent applications. It also becomes easier to integrate IPv6 computers in IPv4 network. The aim of such addresses is to facilitate the transition towards IPv6.

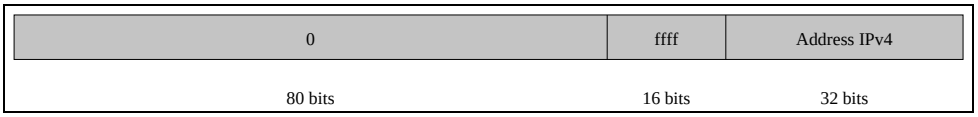

Figure 2: IPv4 mapped address format: the 32 last bytes are the IPv4 address

Multicast addresses. This type of addresses, illustrated on figure 3, are used in accordance with the multicast protocol, one of the new features in IPv6. Such an address represents a group, which is a set of interfaces. The first byte is constant: *ff*. The second field is *1* if the address is temporary and *0* in the other case. The third field gives the scope of diffusion. For example, *1* is for a node-local scope and *2* for a link-local scope. The following bytes are the group ID.

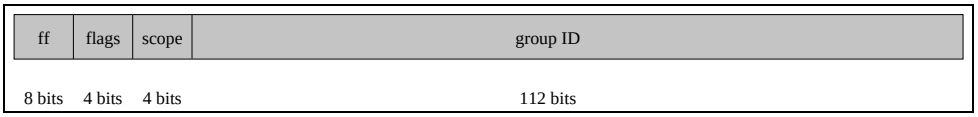

Figure 3: IPv6 multicast address format

The principle of multicast which was already experienced under IPv4 with the mbone[3], has now been completely integrated in IPv6. We will discuss on it in section 2.4. It is interesting to notice that the broadcast address does not exist any more; it has been replaced by multicast addresses. All details concerning the IPv6 addressing architecture can be found in [12].

2.2 Protocols

Two important principals guided the definition of IPv6 packet headers:

[3] `http://www.mbone.com`

- a simple, minimalist approach should be taken, avoiding lots of statically allocated but rarely used option fields;
- the architecture should be extensible, allowing the easy addition of optional features. For all its simplicity, even the basic form of an IPv6 header offers some important new features.

2.2.1 IP

The modification of the address length is not the only change that occurred in the IP protocol. The aim behind all the modifications is to optimize the treatment inside a router in order to reduce the latency in each intermediate router along a packet's delivery path.

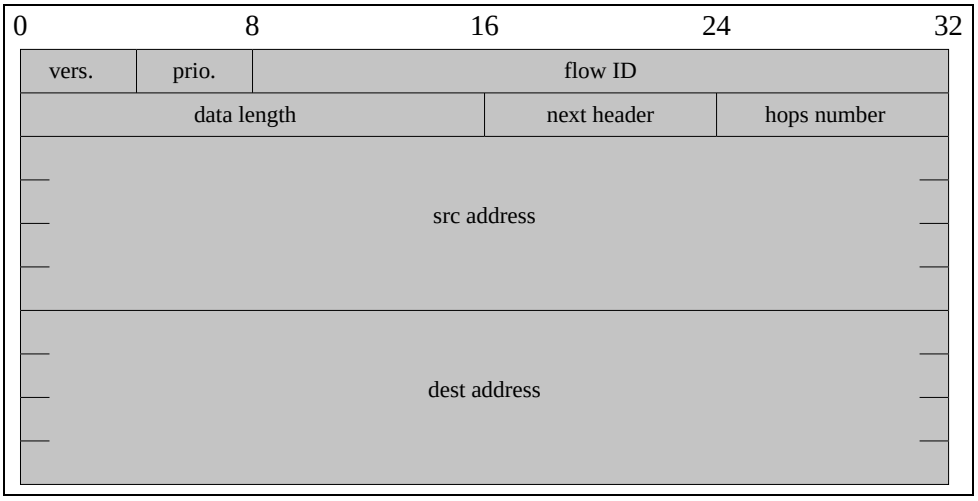

Figure 4: IPv6 header fields

The most important optimization is the simplification of the header format which has now a fixed size. The new header is illustrated on figure 4. Some IPv4 header fields have been dropped or made optional to reduce the common-case processing cost of packet handling and to keep the bandwidth overhead of the IPv6 header as low as possible in spite of the increased size of the addresses

The header does not contain any more the checksum field. Indeed, because of the decrease of the TTL (*Time To Live*) field in the IPv4 header at each hop, (now the hop value), the checksum field had to be computed by all routers receiving the packet which yield to memory copies and waste of computation time in each intermediate router. For not loosing any security, the verification mechanisms are now implemented in higher-level protocols. They all must have a checksum which includes a pseudo-header to verify that the IP characteristics

have not been changed. For example the UDP checksum which was optional under IPv4 is now mandatory.

The size of the header is fixed. There are no more options in the header. The aim here is to increase and optimize packet processing inside each routers. Indeed, it is much more easier to treat fixed size headers than variable ones. The options have been replaced by new headers called *extensions* which play quite the same role. Fields are aligned on 64-bit words. The minimal MTU (*Maximum Transport Unit*) size has been set to 1280 bytes. The *hops number* is a value which is decreased by every router. When it reaches *0*, the packet is dropped and an error message is sent to the originator of the packet. The *traffic class* and *flow ID* fields will be discussed in section 2.4.

One important point is that fragmentation does not exist any more. It has become useless with the use of a *Path Maximum Transport Unit* discovering algorithm. This algorithm works in the following way. First it decides that the PMTU is equal to the MTU of the local link. If the packet is too long, an error message is received and the supposed PMTU is decreased. Knowing that it cannot be under 1280 bytes, the PMTU is found. During a connection, longer packets can be sent in order to verify that the PMTU has not increased. Since this protocol works on the *too big packet* ICMP error message, higher protocols have to provide some reliability and eventually to retransmit data.

Extensions are a new concept of IPv6. It is quite similar to IP-in-IP encapsulation. They are headers that are added to the IP header, before or after, depending on the *extension*. They provide particular informations for particular circumstances. For example, it could be an *ESP header* (ESP is presented in section 2.3) to give informations about the attributes of the security association used with the packet, or an *IPv4-tunneling header* if the packet is encapsulated in IPv4. A key extensibility feature of IPv6 is the ability to encode, within an extension, the action which a router or host should perform if an option is unknown. This permits the incremental deployment of additional functionalities into an operational network with a minimal danger of disruption. All details concerning the IPv6 header format are given in [8].

2.2.2 ICMPv6

ICMPv6 is more than a simple evolution of ICMP. It now also supports the functionalities of ARP and IGMP.

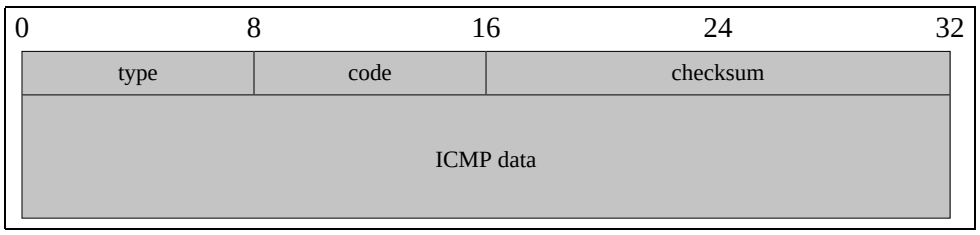

Figure 5: ICMPv6 header fields

ICMPv6 still handles the control or error messages like *unreachable destination*, *no route to host*, *packet too big*. The type of the message is set in the *type* field of the ICMPv6 packet header as illustrated on figure 5, and, if necessary, a precision is given in the *code* field. But ICMPv6 is also responsible for the *neighbor discovery* protocol. This protocol is used for different tasks:

- *Address Resolution*: the principle is quite the same as for ARP in IPv4. The difference is that no new protocol is defined; it is handled by ICMP.
- *Neighbor Unreachability Detection*: this feature did not exist under IPv4. Its aim is to erase unreachable hosts in the configuration tables and to change the routing table if a router fails down.
- *Configuration*: the automatic configuration of the computer is another new feature of IPv6. It gives the possibility to a newly connected computer to determine its addresses (in fact the prefix), routers, duplicated addresses and different parameters such as the MTU of the link, the maximal hops number.
- *Redirection Indication*: this message is used when the router knows a better route.

IGMP, the level 3 protocol which was used under IPv4 to manage multicast sessions, does not exist any more. Its particular tasks are now handled by IGMPv6. The different messages and the multicast protocol are discussed in section 2.4. All details concerning ICMPv6 are given in [7].

2.2.3 TCP and UDP

Modifications made to TCP and UDP concern two main areas. The first one deals with data integrity: a pseudo IP header has been added to the computation of the checksum. In UDP, its computation becomes mandatory and it is integrated in the header as it was in IPv4. For both protocols, it must contain a pseudo-IP-header, which is depicted on figure 6.

The second one concerns packet length. It is now possible to send *jumbograms*[2], which are packets longer than 65536 bytes. They can notably be useful in distributed computation. For example, an IP over Myrinet will benefit from the introduction of jumbogram. Since there is no physical MTU fixed in Myrinet, packet can be of any size which allow to achieve efficient throughput.

2.3 Support for authentication and privacy: IPsec

IPv6 includes the definition of an extension which provides support for authentication and data integrity. This extension is included as a basic element of IPv6 and support for it will be required in all implementations. IPv6 also includes the definition of an extension to support confidentiality by using encryption.

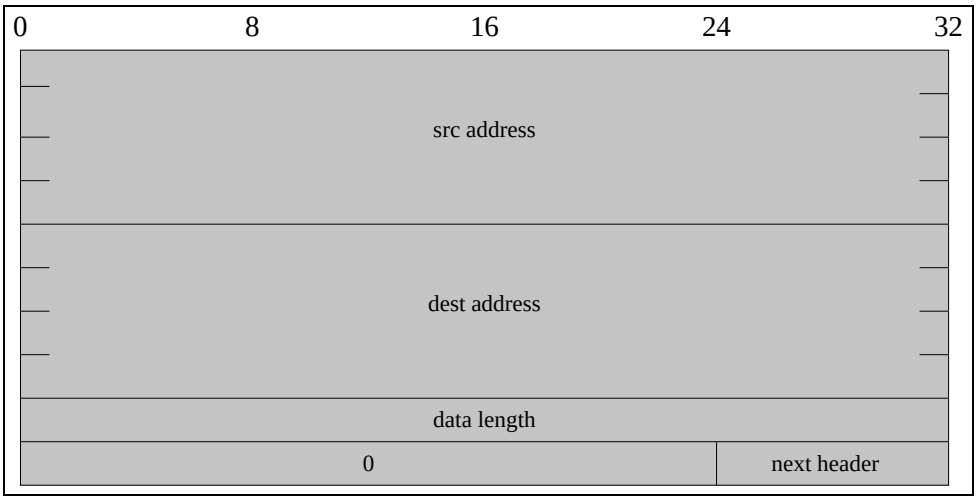

Figure 6: IPv6 pseudo-header for checksum computation

Two long-sought security options - offering security features at the router level of the TCP/IP architecture, where they can benefit all TCP/IP applications - have already been defined as extensions to the IPv6 header.

The IPv6 Authentication Header prevents unauthorized hosts from sending traffic to certain destinations by first forcing the sender to "log into" the receiver in a secure way. This extension leaves the specifics of the authentication algorithm up to the implementers, but promises to eliminate a significant class of network "hacker" attacks.

To prevent the interception of sensitive traffic, the IPv6 Encapsulating Security Header allows the IPv6 traffic exchanged between two hosts to be encrypted. Also algorithm-independent, its standard use of DES CBC encryption will mean secure traffic even through the Internet.

Security is such an important issue that you may not need to wait for IPv6 to get it. The IPv6 security specifications provide encapsulation of the IP security payload in IPv4, essentially emulating the Next Header Payload field of IPv6. However, while this implementation is an optional issue in IPv4, it is a mandatory part of IPv6.

2.3.1 The use of IPsec

IPsec, *IP Security Protocol*, designs a set of mechanisms, at the IP level, which aim to provide the following security services:

- Confidentiality: it protects data from being read by non-authorized individuals;
- Integrity in non-connected mode: it protects data from being changed during the transfer;

- Authentication: it gives the proof that the received datum is from the declared entity;
- Protection against replay.

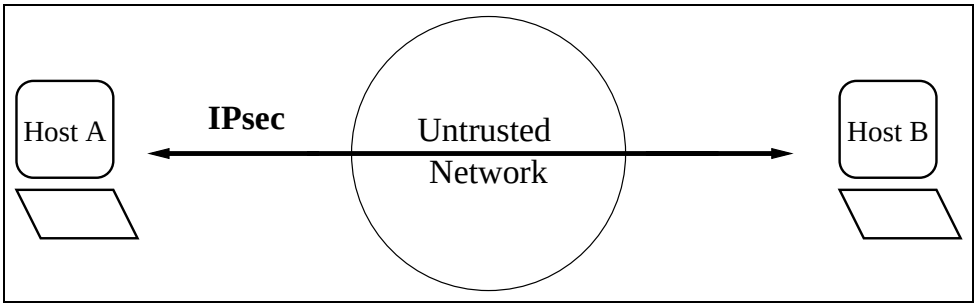

Figure 7: The host-to-host configuration

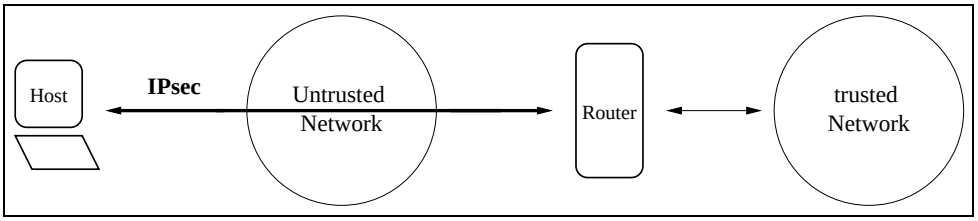

Figure 8: The host-to-router configuration

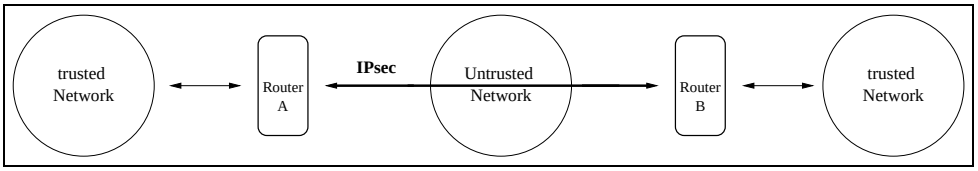

Figure 9: The router-to-router configuration

IPsec is a network layer protocol, which means that it provides its services to every higher protocols in a transparent way. Another point of interest is that it can not only be used for end-to-end security but also to protect networks.

The three main situations for which IPsec is used are the following: host-to-host, host-to-router, router-to-router. The first configuration, presented in figure 7, takes place when two hosts want to communicate over a non trusted network. The second one, presented in figure 8, occurs when there is a need to provide secure accesses to local networks for mobile

interfaces. The figure 9 illustrates the third case which is the creation of a VPN, for *Virtual Private Network*: two private networks need to communicate over a non secured one.

2.3.2 Architecture of IPsec

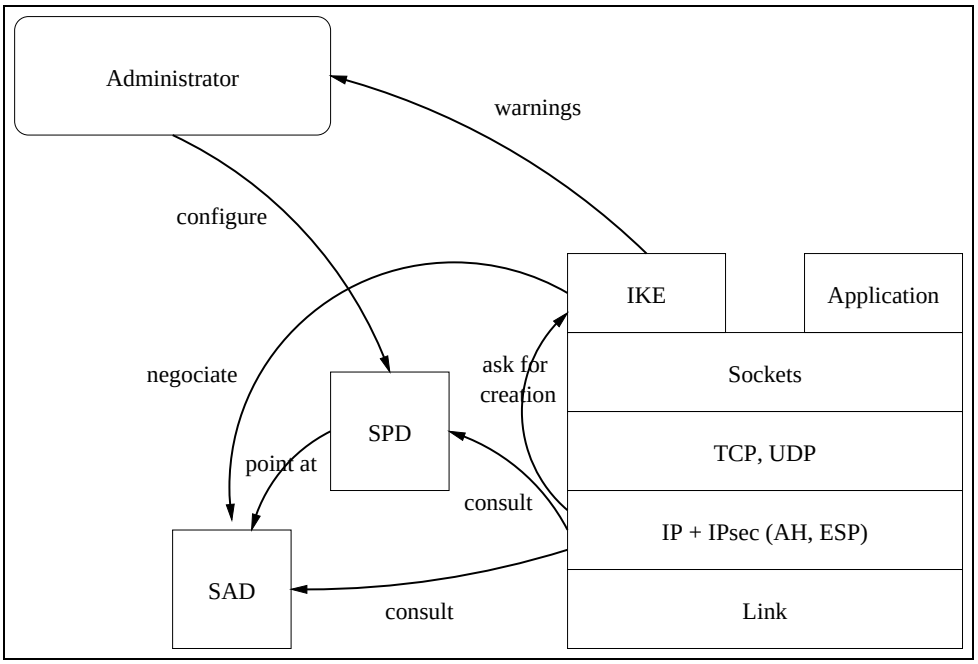

Figure 10: Architecture of IPsec

To provide its service, IPsec uses two protocols: ESP, *Encapsulating Security Protocol*, and AH, *Authentication Header*. ESP's aim is to provide confidentiality and authentication by the use of strong cryptography, while AH only provides authentication and integrity. ESP and AH make an intensive use of extensions, presented in section 2.2.1, to transmit necessary information.

ESP and AH rely on cryptography and some parameters must be shared by the two hosts for their processing. To manage these parameters, IPsec introduces the notion of SA or *Security Association*. A SA is a set of parameters which give all the information necessary for the use of IPsec. A SA is unilateral, which means that it only concerns a simplex connection. To provide total security on a duplex connection, two SAs must be set. A protocol has also been developed to process SA negotiations: IKE, *Internet Key Exchange*. When a new SA is needed, in order to protect a new connection for example, an IKE daemon is used to dynamically negotiate the desired SA with the other host, and record it in the SAD, the *Security Association Database*.

To know in what conditions IPsec must be used, SPs, *Security Policies*, are defined. A SP tells how outgoing as well as incoming packets must be managed. SPs are recorded in a SPD, a *Security Policy Database*, which is managed by the system administrator. The global architecture of IPsec and the interactions between the SPD, the SAD and the IP stack are illustrated in figure 10. For more informations about it, see [14].

2.4 Other new features

Therefore, multicast is mandatory in every IPv6 hosts/routers whereas it was optional in IPv4. The principle of multicast opens numerous perspectives in the domains of meta-computing or distributed computation. Indeed, it enables the dispatch of information and communication between several entities in a new and efficient way and appears to be a key point in Distributed Interactive Application (DIS) [15].

2.4.1 Multicast

With IPv6, multicast addressing is likely to find wider use. IPv6's new multicast address format allows for trillions of possible multicast group codes, each identifying two or more packet recipients. The scalability of multicast routing is improved by adding a "scope" field to multicast addresses in order to confined a particular multicast address to a single system, restricted within a specific site, associated with a particular network link, or distributed worldwide. Multicast provides more flexibility in a more efficient way and it is heavily used even in the most basic protocol like Neighbor Discovery Protocol (NDP).

The aim of multicast is to deliver a same packet to a set of interfaces. The set of interfaces waiting for the packet is called a group. Management of the group is the purpose of ICMPv6. To achieve this task, three types of messages have been defined.

- the first is regularly sent by routers to know if some interfaces are still interested in receiving packets;
- the second is sent by interfaces which want to join a group or as an answer to the first message;
- the third message is sent when an interface wants to leave a group.

Finally, IPv6 introduces a new kind of address: an anycast address. An IPv6 anycast address is a single value assigned to more than one interface, typically belonging to different computers. A packet sent to an anycast address is routed to the "nearest" interface having that address, according to the measure of distance used by the routing protocol.

2.4.2 Mobility

The principle of mobility is quite new. The aim is that computers can stay connected to the Internet despite their physical moves. It assumes that a computer can automatically obtain an address when it is connected to an IP network but also that it is reachable on a

constant address. The result is a great flexibility in network management and architecture. Here is an example. *Host A* sends a packet with the *mother address* of *host B* as destination address. The router of the *mother network* of *host B*, with which *host B* did an association redirects the packet towards the *temporary address* of *host B*. A protocol, based on the use of *extensions*, was written to negotiate such associations (between the *mother address* of a computer and the router where it is actually located). This new capability may allow new interfaces to be easily designed to support a range of scientific activities across distributed, heterogeneous computing platforms [16].

2.4.3 Quality of service capabilities

Concerning routing, the IPv6 header introduces two important "Quality of Service" features to the Internet Protocol: flow labels and traffic class.

A flow is a sequence of packets sent from a unicast source to a unicast or multicast destination. The IPv6 flow label enables the flow's source to identify a logical sequence of packets; intervening routers that support this feature can then maintain a context for the flows currently in transit, thus opening the door for possible optimized performance and congestion management - resulting in faster, more reliable networking for everyone, all quite impossible with IPv4. Thus the useful field *flow ID*, chosen by the source can be used by routers as a context ID, for example to increase the speed of packet processing. Indeed, routers do not need any more to watch after 5 fields but only one to determine the context.

Communicating hosts can also specify a packet priority, a feature that will allow IPv6 routers to discriminate and favorably accommodate TCP/IP applications that require faster response time (for example, interactive applications like TELNET, SNMP network management traffic...). This feature also opens the door to a host of possible real-time network applications that could never have been implemented with IPv4, and may enhance distributed applications that have specific features like DIS [4]. The appearance in the IP header of a field called *traffic class* enables a differentiation of services. The principle is to give to access providers the freedom to handle differently network congestion. Without differentiation, each packet has the same probability to be dropped. But with differentiation, several classes are defined. Packets with higher classes will have a higher probability not to be refused. The main interest of this model is that it does not break the concept of "Best Effort" used in the Internet. A packet of high class is not sure to reach its destination; the probability is only higher. Of course each provider would have to set a policy in order that the highest class does not become the only one used. For a more precise description of IPv6, see [6].

3 An IPv6 networking package

The first step of our work, bringing IPv6 to Java, was to create the API that would give to programmers the possibility to deal with IPv6 networks and to benefit from all the new functionalities that we mentioned before. This API had to be coherent in regard with the

Java architecture and, in particular, with the already existing network programming API, which is described in the Java documentation[4].

Network programming under Java is made through the use of the *java.net* package. It contains classes which deal with enough mechanisms to create advanced network applications. To design our package, called `fr.loria.resedas.net6`, we decided to start from this original package.

3.1 Architecture

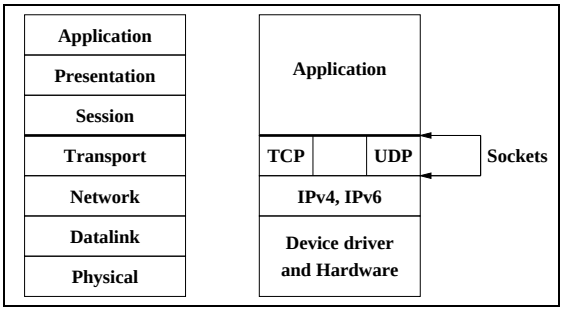

Figure 11: The OSI model and the TCP/IP protocols

If we look at the different classes of the *java.net* package, three main levels can be discerned. These three levels are depicted on figure 11. The first one is composed of classes which only deal with Java mechanisms: the *exceptions*. The second one is composed of classes which implement session or application mechanisms, like *URL*, *HttpURLConnection* or other *ContentHandler*. Finally, the third one deals with transport mechanisms and roughly speaking with TCP/IP. We will call it the *networking set* and we will work at this level since this is typically the right place where we need to plug IPv6 functionalities. The classes belonging to the networking set are listed in figure 12.

+ DatgramPacket	+ ServerSocket
+ DatagramSocket	+ Socket
+ MulticastSocket	+ SocketImpl
+ DatagramSocketImpl	+ SocketImplFactory
+ InetAddress	+ SocketInputStream
+ InetAddressImpl	+ SocketOptions
+ PlainDatagramSocketImpl	+ SocketOutputStream
+ PlainSocketImpl	

Figure 12: The networking set

[4] `http://java.sun.com/docs/index.html`

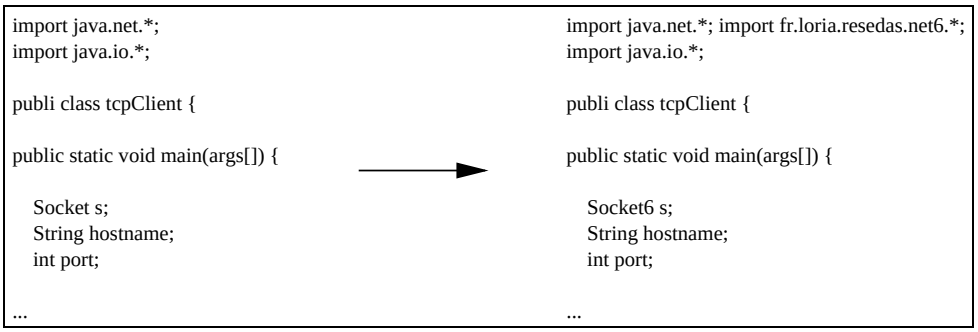

Figure 13: From an IPv4 to an IPv6 program

+ DatagramPacket6	+ SecurityManager6
+ DatagramSocket6	+ Socket6
+ DatagramSocketImpl6	+ SocketImpl6
+ InetAddress6	+ SocketImplFactory6
+ InetAddressImpl6	+ SocketInputStream6
+ MulticastSocket6	+ SocketOptions6
+ PlainDatagramSocketImpl6	+ SocketOutputStream6
+ PlainSocketImpl6	

Figure 14: The *fr.loria.resedas.net6* package

The *networking set* captures all the links between the Java world and the network stack. The only way for an object to interact with the network stack is to take an object of this class as peer. This means that we have a model where a modification to a class of the *networking set* will modify the whole behavior of Java with network. Furthermore, this modification may be transparent for the upper-layer objects if the *networking set* interface is not modified.

In order to provide a good integration ability to our package and to make the development of upper-layer objects easier, we consider the structure of the *networking set* and decide to keep the same structure. The composition of the *fr.loria.resedas.net6* package is illustrated on the figure 14.

The first obvious advantage of this choice is code reuse. Not only for the design (and coding) of the package itself but also for network interacting objects. The transition from already existing IPv4 objects, towards an IPv6 networking ability would be simple, flexible and easy to perform. In most cases, it can be automatically done by syntactic transformation (we used `sed`). A short example is illustrated in figure 13. A second benefit is that we did not introduce any new classes. Thus, the underlying semantic of Java is not changed, the security is not depreciated and in a more practical way, network programming with Java remains the same. The last but not the least advantage is the facility of integrating such a package in a JVM. We will see in details in section 5.4 what this really means.

3.2 Derivation from the *java.net* package

The first decision we had to take in the translation of the networking set from IPv4 to IPv6 was a purely syntactic one. We have to find new names and then upgrade all the references off the concerned objects. This step was performed automatically by using a simple stream editor. Indeed modifications were all minor, and consisted of, for example, changing *Socket* in *Socket6*.

In order to handle IPv6 specifications, like the storage size of addresses, the other major modification was focused on the internals of objects. The main change was for the *InetAddress* class. Indeed, in Java, an IPv4 address is stored in the *address* field of *InetAddress* objects, whose type is an `int`. This is not enough to store IPv6 addresses. So we changed the type of this field for a `byte[16]` one. This modification led to other ones, concerning all aspects of address representation, address management...

Concept changes between IPv4 and IPv6 were also responsible for some other modifications. For example, the management of addresses by the *DNS cache* stored in the *InetAddress* class had to be modified. In the original version, when a call to the *InetAddress.GetHostName()* method is performed, the current IP address is compared to already known addresses for this hostname. If some differs, the hostname is hidden to the user and the pair address and hostname is not cached. This behavior is no longer acceptable under IPv6 where a host has frequently several addresses, depending on its interfaces and its connectivity providers. In our package, the user gets the hostname and the pair is cached.

Of course, we did not limit ourselves to the already existing interface but improved it by integrating new IPv6 features. To introduce new options provided by IPv6, we enriched some objects with new *methods*. However, the design of these functions was made accordingly to the similar functions of IPv4. More important, the link that they introduce between the different layers of the package are exactly the same as for those of the original package. This hierarchic structure between objects used by programmers and the underlying library is important for several points.

The first is the control of accessible resources from the upper-layers. An example is the *socketSetOption* method of the *PlainSocketImpl* class. In theory, this method gives the possibility to set any option for the concerned socket; this is the equivalent of the *setsockopt* system call in C. However this method is not accessible from the upper-layer. It has been split in several other methods of the *Socket* class such as *setTCPNoDelay* or *setTimeOut*...

A second point is the extensibility of the system. If a new option is introduced, the managing method, the one which makes the link with the underlying library, already exists. The only work would be to introduce a method in the higher level, to make the option accessible.

3.3 The security manager problem

Keeping the interface of the *networking set* in our package was not always possible. For example, the return value of the *ServerSocket.accept* method is an object of the class *Socket*. It is necessary that in our package, the return value of *ServerSocket6.accept* is an object of

the class *Socket6*. Those changes had no consequence as long as the considered methods belong to classes of our package. When this is not the case, compatibility is no more assumed. We accept this incompatibility when the objects to which the method belongs are objects which deal with network at a higher level, as in the case of the *URL* class. The aim of this package is not to reproduce the whole network features of Java but only to give enough tools to deal with IPv6.

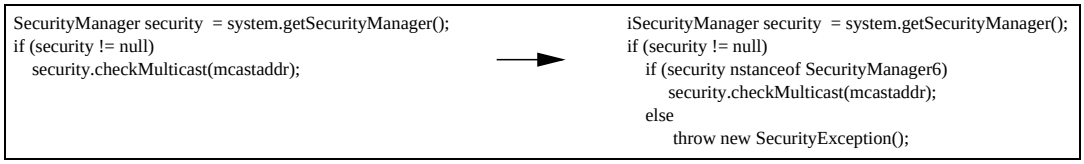

Figure 15: The *SecurityManager* problem

The real problem was for the *Security Manager* class. Two of its methods, *checkMulticast*, take objects of the *InetAddress* class as arguments. This was embarrassing because no *checkMulticast(InetAddress6)* was available. Refusing to create a security hole by not testing the *Security Manager*, we dodged the problem. We designed a *Security Manager6* class, inheriting from the previous and updated with the desired methods. Then, when asking for multicast, we previously test if the loaded *Security Manager* belongs to the *Security Manager6* class. If this is the case, we call the methods; if not, we throw a *SecurityException*. The changes in the code are depicted on figure 15.

4 The underlying mechanisms

Under Java, the development of new features, outside the standard Java class library, requires to deal with mechanisms that are themselves not available from the Java™ Virtual Machine. In our case, these mechanisms deal with operations on IPv6 sockets and addresses. In order to be able to perform such operations, we had to develop a library which would embed the Java™ Virtual Machine into native network operations. Before entering in more detail the implementation of the library, let us give some informations about JNI.

4.1 JNI

In order to build our library we use the *Java™ Native Interface* (JNI). JNI allows Java code that runs inside a JVM to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. The interesting characteristics of that interface is that all the Java world is visible from the C code. This means that we can, for example, manipulate Java objects, invoke methods and throw exceptions directly with C calls. The bad point is that the safety in the Java world is no more assumed. Consequently, a great attention is required. The principal critical points of JNI are the management of exceptions and memory.

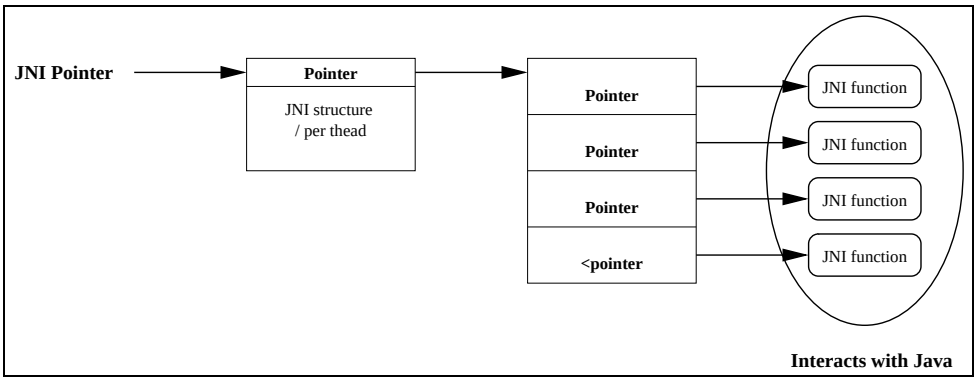

Figure 16: The link between C and Java

The internal functioning of JNI is illustrated on figure 16. A C pointer, the JNI pointer, points towards a structure. It is composed of an environment and another pointer. For each thread, an environment is defined. The pointer points towards a collection of function pointers which are the JNI specific functions. Their aim is to interact with the Java world.

The use of JNI is quite simple. The first step consists in declaring the native methods in their corresponding objects without defining them. The key-word *native* is used to tell the compiler that they are native methods. The second step is the writing of the library in C or C++. Calls to native methods are the same as for classical ones. For more information about JNI, please refer to the web pages[5].

4.2 Implementation

The library we wrote is composed of all the native functions which had been declared in our package. The writing, in C, of these functions could be split in three steps. If the first one could be seen as a normal system code writing, the two others are a consequence of the interactions between Java and C through the use of JNI

The first one was the writing of the effective part of the code. By effective part we mean the part that performs the actions in relation with sockets and Internet addresses. In a global way, they are the system calls. However, if we are more precise, it is not exactly like that. Indeed the necessity for our library to be *thread safe* led us to a particular work which will be explained in section 4.3. For more information on network programming under Unix systems, see [18].

Managing the Java objects was the second step. Under JNI, access to Java objects is made with the help of a particular variable, the environment. The manipulation of this variable cannot be done directly but only through the use of a set of specific JNI functions.

[5] `http://java.sun.com`

Because of that phenomena, every action on a Java object is embedded by calls to functions used to access and modify it.

The last step, but not the least important, was the management of errors. First of all the code had to be protected against runtime crashes. A crash during the execution of the library would lead to a JVM's crash. Indeed, an error in the C world cannot be interpreted by the JVM and so cannot be handled. The main result is that the reliability of Java programs which compiled and therefore should not crash, is no more assumed. Then it was necessary to translate every caught error of the C world, for example the values of the *errno* variable, in Java exceptions.

4.3 Thread safety

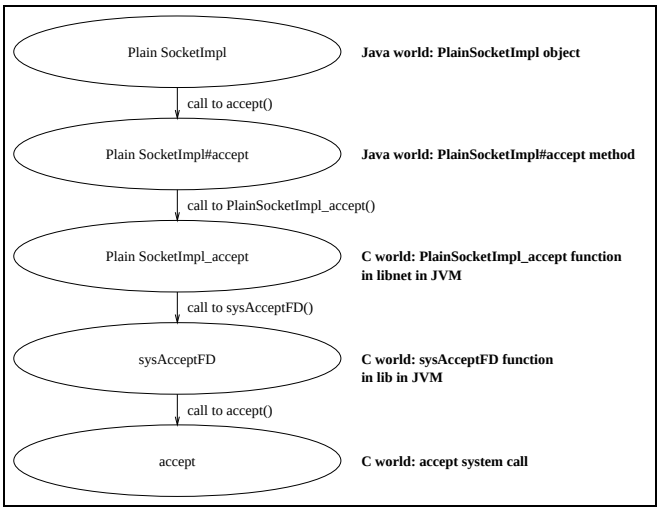

Figure 17: The level of the library in JdK 1.1

```
struct fddescriptor { int fd; }

typedef struct fddescriptor Classjava_io_FileDescriptor;

extern int sysSocketInitializeFD(Classjava_io_FileDescriptor *, int);
```

Figure 18: An exemple of intermediate structure (JdK 1.1)

The main problem we encountered was the question of thread safety. Working in Java induces working in a multi-thread environment, and such an environment requires to take some precautions. Without being *Java-thread safe* the library could not be effective. This is the reason why we could not use classical system calls in the library without any particular precautions. The complexity of the problem is even higher because we cannot make any assumption on the kind of threads that are used during execution. They could be native threads or green threads and we do not want to depend on it

In fact, only a limited number of system calls were concerned. In particular, they are mostly I/O calls like *open*, *read* or *write*. The important point is that all these calls are already used in the JVM, at least to support the *java.net* package. Thus, the *thread safety* problem has been already addressed. In the JVM, the controversial system calls are mapped in what we will call pseudo functions which take all precautions in connection with threads. Figure 17 depicts how a system call like *accept* is mapped to a pseudo function named *sysAcceptFD* and illustrates the final structure of the code.

The internal of these functions is not important for the purpose of our discussion. The most interesting point is that since this pseudo functions were already implemented for the system calls in question, we decided to integrate our library into the architecture of the JVM by proceding the same way as the classical network library. Our C functions would call the same pseudo functions. For more information about network programming, please refer to [17].

While interacting with these pseudo-functions was not a problem with JdK 1.2, we encountered a difficulty in the case of JdK 1.1. Indeed, the arguments the JVM functions (JdK 1.1) wait are argument structures which were defined in a previous version of JNI. These structures are no more available with the version we used. So we had to create intermediate structures that we could manipulate and that are compatible with the JVM library functions. We qualify them intermediate because they are not a reproduction of the structure that is needed by the JVM functions, but a dummy with enough informations to satisfy them. For example, the *sysSocketInitializeFD* JVM function wait for an argument of type *Classjava_io_FileDescriptor* but only deal with the *fd* field of this argument. So, the intermediate structure we created is composed of a only *fd* field. Figure 18 illustrates the declaration of the structure.

5 Results and extensions

After having written the IPv6 Java API as well as its underlying library, we tested it. Even before results, the first fact that merged out of the experiments was that our package did not take advantage of all the new features that IPv6 provides, the same ones that motivated for a great part the development of the technology. The last step in the development of the package was the introduction of specific features provided by IPv6.

5.1 A raw level

In a classical Java environment, raw sockets are not accessible. This could represent a lack of flexibility in several situations. For example, one could want to use *emulators* to reproduce particular environments for testing purposes and thus generate specific traffic (ICMP, router alert...). We could also need to monitor network traffic when using or optimizing parallel applications [1]. For these purposes, we wrote a raw socket interface for Java.

The raw interface provides several objects that could be used, whether to generate ICMPv6 packets, or IP packets. The set of objects which deal with the raw level is quite the same as the one for UDP sockets. We defined the following objects which play the same role as their UDP counterparts.

- RawPacket6
- RawSocket6
- PlainRawSocketImpl6
- IcmpPacket6
- IcmpSocket6

Of course, the whole package is not dependent on the presence of the raw objects. If the introduction of this interface is not acceptable for the network security, it could be easily removed. Moreover, it is necessary to have *superuser* rights to access the raw package ressources.

5.2 New options

The creation of a package for networking under IPv6 would have been a non-sense without the addition of the new features specific to that protocol. New IPv6 options were also added to the network API. They concern multicast, IPsec and flow control.

Multicast. We described it in section 2.4, and it already existed in Java. The interface provided by our package is the same as the one of the *networking set*. There are only two methods whose name has changed: *setTTL* has become *setHops* and *getTTL* has become *getHops*. Otherwise, the functionalities and the way to program and use multicast are the same.

Authentication and privacy. For IPsec, which is quickly presented in section 2.3, things are different. Though IPsec was defined under IPv4, it had never been included in a Java package. In the *fr.loria.resedas.net6* package, it is possible for an object to ask for a particular security policy to apply on a socket; of course, it is only possible if the IPv6 stack implements this option. The interest of IPsec lies on the simplicity of the mechanisms; no configuration is necessary. The object does not have to settle a particular protocol or to

agree with a peer. It just has to set the security level it wishes. For example, consider the program presented in A. It is a little echo client taken from [10] and modified to take advantage from our IPv6 package. After the creation of the socket, the programmer sets the ESP Security Policy, which is introduced in section 2.3.2, for this socket to 5. With the stack developed by Kame[6], it means that each packet transiting through this socket will be encrypted.

Quality of service capabilities. The flow control is another new thing included in the package. It relies on two mechanisms that are described in section 2.4. The API is the following: through the use of several new methods, *setFlowLabel, setTrafficClass, getFlowLabel* and *getTrafficClass*, it is possible to ask for a particular traffic class and flow label to apply on a socket. For example, in our echo client, the flow label and the traffic class of the socket are set to 0. Unfortunately, no API has been specified yet to effectively set these values in the IPv6 header. Therefore, the Java interface we provide has no real effects. This will be fixed as soon as an API will be defined.

5.3 Results

After having written the package and the library, we planned to translate several Java applications from IPv4 to IPv6. As we already said it, that was made using a stream editor, the only differences being syntactic. We began from small applications like secured telnet clients or ping programs to finish with greater ones.

To give an example, we could consider the software called *jmrc*. It is a Java multicast chat program. This program is interesting because it is a good example of the resources that our package can provide. We have been able to run it and it perfectly executed under IPv6. We also were able to partially protect the communications between the users. The flow control could not be tested yet. For more information, please refer to the web page of jmrc[7].

5.4 An IPv6 compliant JVM

5.4.1 Principle

The good behavior of our IPv6 package opens new perspectives. Instead of just adding a new package to the JVM, we decided to completely replace the IPv4 network mechanisms present inside a classical JVM by the IPv6 one described above, in order to have a transparent IPv6 compliant JVM. The architecture is depicted on figure 19.

The manipulation consisted in modifying the internal of the *java.net* package without modifying names, *i.e.*, doing a perfect matching between IPv4 methods of the old *java.net* and IPv6 methods of our *java.net6* package. The modifications were the same as those we had performed on the *fr.loria.resedas.net6* one. In this case it was even easier because

[6] `http:///www.kame.net`
[7] `http://www-res.enst.fr/~dax/guides/multicast`

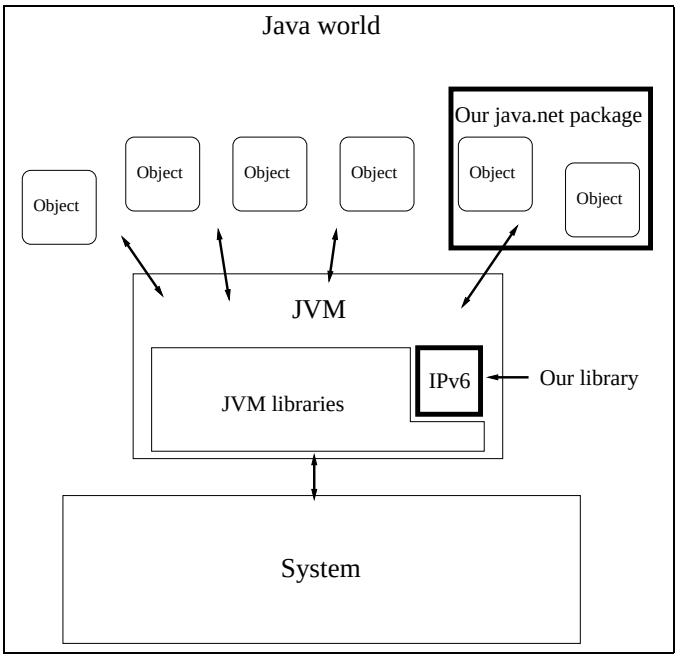

Figure 19: The global architecture

fewer problems in compatibility were met, the interface of this modified package being very similar. Our library required only few modifications on a syntactic level to work.

5.4.2 An IPv4/IPv6-able JVM

After replacing the original *java.net* objects with ours and integrated the IPv6 library in the internals of the JVM the result was completely satisfactory with the following main interesting points:

- We have now a JVM under which every Java program can communicate under IPv6 without any modification.
- Development of new softwares is exactly the same for IPv6 as for IPv4.
- Any Java objects can not only communicate under IPv6 without any modifications, but also manage IPv4 messages. Indeed, IPv6 provides simple and flexible transition from IPv4. The key transition objective is to allow IPv6 and IPv4 hosts to interoperate. The Simple Internet Transition (SIT) specify for example a model of deployment where all hosts and routers upgraded to IPv6 in the early transition phase are "dual" capable (*i.e.*, implement complete IPv4 and IPv6 protocol stacks) In other words, SIT ensures

that IPv6 hosts can interoperate with IPv4 hosts anywhere in the Internet. This means that our JVM allows any Java program to work under IPv6 as well as under IPv4.

- In particular, any Java application using networking will work in a IPv6 environment by using indistinctly IPv4 addresses or IPv6 addresses and becomes able to performed *Remote Method Invocations* (RMI) on a IPv4 JVM or in a fully IPv6 network environment.
- Modifications made to the JVM are not irreversible. Indeed, it is very easy to switch between an IPv4/IPv6 JVM and the classical IPv4 JVM. It is even possible to specify the one to use at the launch of applications.

To validate the IPv6 compliant JVM, we run several unmodified Java applications on a dual stack platform. The experiments show that, except for IPv4 Multicast, both protocols are fully supported. This means, for example, that telnet servers are now accepting IPv6 and IPv4 connections. The multicast compatibility problem is described in 6.1. Tests of RMI were also performed and were conclusive since RMI clients an servers now support IPv6 networks. As main example of tested application, we could consider the *Jigsaw* web server which is now running under IPv6 without code modficications. For more information about this software, please refer to the web pages of Jigsaw[8].

6 Last updates and availability

In this section, we present the last works that have been achieved on several problems that were met during the testing of our package. We also give the availability of our package: the different OS and stacks that are supported and the limits of each versions.

6.1 Last updates

Some problems related to IPv6 are still not solved and several groups are still working on different issues. Among all of them, two were encountered during the testing of our package: they are related to compatibilty with IPv4 and URL syntax. We briefly present here the work that was achieved to solve them, by our team as well as other groups, and their impact on our package.

6.1.1 IPv4 Multicast under IPv6

Dual stacks. To provide a simple and flexible transition from IPv4 to IPv6, an interoperation between IPv4 and IPv6 hosts is needed. The Simple Internet Transition (SIT) solves this problem by specifying a model of deployment where all hosts and routers upgraded to IPv6 in the early transition phase are "dual" capable *i.e.* own dual stacks.

[8] `http://www.w3.org/Jigsaw/`

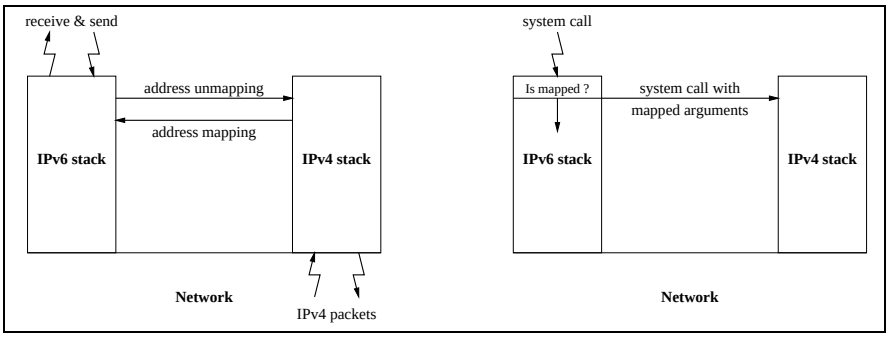

Figure 20: Basic working of a dual stack

A dual stack is a stack that implements both IPv6 and IPv4 protocols. Moreover, it gives the ability to IPv6 sockets to deal with IPv4 packets, by using IPv4-mapped addresses that are described in section 2.1. Basically, the working is the one described in figure 20: when an IPv4 packet is received, it is first processed in the IPv4 stack and then passed to the IPv6 socket with the corresponding IPv4-mapped address; when a IPv6 sockets sends a packet to an IPv4-mapped address, the packet is passed to the IPv4 stack with the corresponding IPv4 address for destination.

Mapping of multicast options. The testing of our IPv6 JVM showed that, in order to achieve an efficient compatibility, a mapping between particular IPv4 and IPv6 options was necessary. These options concern multicast and are:

- *IP_MULTICAST_TTL*
- *IP_MULTICAST_IF*
- *IP_MULTICAST_LOOP*
- *IP_ADD_MEMBERSHIP*
- *IP_DROP_MEMBERSHIP*

on IPv4 side and

- *IPV6_MULTICAST_HOPS*
- *IPV6_MULTICAST_IF*
- *IPV6_MULTICAST_LOOP*
- *IPV6_ADD_MEMBERSHIP*
- *IPV6_DROP_MEMBERSHIP*

on IPv6 one. The reason why is that a fully IPv6 applications, *i.e.* an application with only deals IPv6 options, can not, for example, join an IPv4 group even through the use of its corresponding IPv4-mapped address. This operation indeed requires to deal with purely IPv4 options.

A solution to that problem is presented in [5]. It proposes to specify a mapping for those specific multicast options in order to provide IPv6 softwares the ability to deal with IPv4 multicast. Several dual stacks have been accordingly modified, see 6.2 for the availability, and tests have been conclusives. For exmaple, our IPv6 JVM now achieves a full compatibility with IPv4 when running on a modified stack.

6.1.2 IPv6 URL

The modification in the string representation of addresses between IPv4 and IPv6 has responsible for a major problem in the syntax of URLs. Indeed, the new form of addresses is no more compatible with their original definition. The use of symbol ":" to represent IPv6 numerical addresses as well as the introduction of a condensed representation for these addresses have introduced an ambiguity in the reading of an URL. For example, the string `3ffe:2c0:20:1::4796:23` can be seen as a numerical address or as an URL composed of address `3ffe:2c0:20:1::4796` associated to port `23`.

To solve this problem, it has been envisaged for a while, to restrict the definition of IPv6 URL to only host names. However, recently, a new standard has been accepted. It is presented in [11]. This new definition makes use of the symbols [and] as addresses delimiters. For example, the string `[3ffe:2c0:20:1::4796]:23` corresponds to the URL composed of address `3ffe:2c0:20:1::4796` associated to port `23`. This new specification has been implemented into the IPv6 compliant JVM. It has required the modification of several classes that do not belong to the *networking set* but rather belong to higher layers or implement security mechanisms inside the Java world. These classes are, for example, *java.net.URLStreamHandler*, *java.security.CodeSource* or *java.rmi.Naming*.

6.2 Availability

The main limitation to the development of the package and the addition of new features is the evolution of IPv6 stacks. Note that our package is running under JdK 1.1 and JdK 1.2. The library has been developed for four of them, under three different platforms.

- Kame[9] for FreeBSD
- Inria[10] for FreeBSD
- Microsoft[11] for Windows NT

[9] `http://www.kame.net`
[10] `ftp://ftp.inria.fr/network/ipv6/`
[11] `http://www.research.microsoft.com/msripv6/`

- Linux: kernel v2.2

The advancement point of each stack is not the same and some APIs are different from one to another. One of the consequences is that some features are only available on certain stacks. For example, IPsec has been, for the moment, only implemented for the Kame stack. Another one is that we had sometimes to modify the code of the library, having to perform different operations depending on the stack in order to reach the same result. A new version of the library is being developed for Solaris.

At this point, the evolution of the package is very dependent on the evolution of the different IPv6 available stacks.

Functionality	OS				
	FreeBSD		Linux	Solaris	WinNT
JdK 1.2 and JdK1.1	INRIA	KAME	2.2.x	v8	NT4 pack4
Basic Ipv6 socket	•	•	•	•	•
Security (IPSEC options)		•			
QoS[a]	•	•	•	•	•
Raw socket	•	•	•	•	
ICMP socket	•	•	•	•	
URL (with IPv6 addresses	•	•	•	•	•
IPv4-mapped Multicast[b]	•	•	•		

Table 1: Summary of IPv6 API functionalities

[a] see remark in section 5.2 concerning the implementation

[b] Modification in the IPv6 stack is needed. Please refer to [5] for further details.

7 Conclusion

We have discussed the design and development of a IPv6 Compliant Java™ Virtual Machine. To the problem we presented in the introduction, development of IPv6 softwares, our work answered in two different ways. The first one, which was the expected one, is the Java package `fr.loria.resedas.net6` which provides all the needed features to manage IPv6 under Java. It integrates the new protocols offered by IPv6 and several new options like IPsec or a raw interface. Its integration needs no modification of the *Java Virtual Machine* and it can be used simultaneously with the classical network resources of the JVM.

The second one is the creation of a JVM working under IPv6. All its network resources have been replaced by ours. The only modifications made to the interface of the classical packages were additions. That means that our JVM assumes an almost total compatibility with already existent softwares. Furthermore, the use of dual stacks assumes that softwares running under our JVM are, in a great part, IPv6 as well as IPv4-able.

These two tools offer new opportunities for the use of IPv6 and for development under Java. Indeed, we offer to programmers the possibility to abstract themselves from the network protocol used, and to network administrators the possibility to grow their IPv6 software resources.

However, progresses can still be made. The problem is that IPv6 stacks are still not exempt of bugs, and that specifications are not always respected. The consequence, for our package, is that portability is not always assumed. Development must still be made in order to really envisage a wide spread of IPv6 around all networks.

A Echo Client

```
 1 import fr.loria.resedas.net6.*;
 2 import java.net.*;
 3 import java.io.*;
 4
 5 public class echoClient {
 6
 7   public static void main(String[] args) {
 8
 9     Socket6 theSocket;
10     String hostname;
11     DataInputStream theInputStream;
12     DataInputStream userInput;
13     PrintStream theOutputStream;
14     String theLine;
15
16     if (args.length > 0) {
17       hostname = args[0];
18     }
19     else {
20       hostname = "localhost";
21     }
22
23     try {
24       theSocket = new Socket6(hostname, 7);
25       theSocket.setNetEncrypt(5);
26       theSocket.setTrafficClass(0);
27       theSocket.setFlowLabel(0);
28       theInputStream = new DataInputStream(theSocket.getInputStream());
29       theOutputStream = new PrintStream(theSocket.getOutputStream());
30       userInput = new DataInputStream(System.in);
31       while (true) {
```

```
32          theLine = userInput.readLine();
33          if (theLine.equals(".")) break;
34          theOutputStream.println(theLine);
35          System.out.println(theInputStream.readLine());
36        }
37      } // end try
38      catch (UnknownHostException e) {
39        System.err.println(e);
40      }
41      catch (IOException e) {
42        System.err.println(e);
43      }
44
45    } // end main
46
47 } // end echoClient
```

B Modifications to the *java.net* API

DatagramSocket6/DatagramSocket

New Methods

- *int getFlowLabel()*: return the flow label that is applied on this socket.
- *int getTrafficClass()*: return the traffic class that is applied on this socket.
- *int getNetAuthenticate()*: get value of the *IPV6_AUTH_NETWORK_LEVEL* option for this socket, *i.e.* the AH policy in tunnel mode to apply on that socket.
- *int getTransportAuthenticate()*: get value of the *IPV6_AUTH_TRANS_LEVEL* option for this socket, *i.e.* the AH policy in transport mode to apply on that socket.
- *int getNetEncrypt()*: get value of the *IPV6_ESP_NETWORK_LEVEL* option for this socket, *i.e.* the ESP policy in tunnel mode to apply on that socket.
- *int getTransportEncrypt()*: get value of the *IPV6_ESP_TRANS_LEVEL* option for this socket, *i.e.* the ESP policy in transport mode to apply on that socket.
- *void setFlowLabel(int)*: set the flow label that is applied on this socket.
- *void setTrafficClass(int)*: set the traffic class that is applied on this socket.
- *void setNetAuthenticate(int)*: set value of the *IPV6_AUTH_NETWORK_LEVEL* option for this socket, *i.e.* the AH policy in tunnel mode to apply on that socket.

- *void setTransportAuthenticate(int)*: set value of the *IPV6_AUTH_TRANS_LEVEL* option for this socket, *i.e.* the AH policy in transport mode to apply on that socket.
- *void setNetEncrypt(int)*: set value of the *IPV6_ESP_NETWORK_LEVEL* option for this socket, *i.e.* the ESP policy in tunnel mode to apply on that socket.
- *void setTransportEncrypt(int)*: set value of the *IPV6_ESP_TRANS_LEVEL* option for this socket, *i.e.* the ESP policy in transport mode to apply on that socket.

DatagramSocketImpl6/DatagramSocketImpl

New Methods

- *int getHOPS()*: get the default hops limit for multicast packets sent out on the socket.
- *void gsetHOPS(int)*: set the default hops limit for multicast packets sent out on this socket.

Removed Methods

- *byte getTTL()*
- *void setTTL(int)*
- *int getTimeToLive()*
- *void setTimeToLive(int)*

InetAddress6/InetAddress

New Methods

- *boolean compare(byte, byte)*: return *true* if the two addresses are equal.
- *boolean isMappedAddress()*: Utility routine to check if the InetAddress is an IPv4-mapped one.

MulticastSocket6/MulticastSocket

New Methods

- *int getHOPS()*: get the default hops limit for multicast packets sent out on the socket.
- *void setHOPS(int)*: set the default hops limit for multicast packets sent out on this socket.
- *void setInterface(int)*: get the multicast network interface used by methods whose behavior would be affected by the value of the network interface. Useful for multihomed hosts.

Modified Methods

- *int getInterface()*: Retrieve the index of the network interface used for multicast packets.

Socket6/Socket

New Methods

- *int getFlowLabel()*: return the flow label that is applied on this socket.
- *int getTrafficClass()*: return the traffic class that is applied on this socket.
- *int getNetAuthenticate()*: get value of the *IPV6_AUTH_NETWORK_LEVEL* option for this socket, *i.e.* the AH policy in tunnel mode to apply on that socket.
- *int getTransportAuthenticate()*: get value of the *IPV6_AUTH_TRANS_LEVEL* option for this socket, *i.e.* the AH policy in transport mode to apply on that socket.
- *int getNetEncrypt()*: get value of the *IPV6_ESP_NETWORK_LEVEL* option for this socket, *i.e.* the ESP policy in tunnel mode to apply on that socket.
- *int getTransportEncrypt()*: get value of the *IPV6_ESP_TRANS_LEVEL* option for this socket, *i.e.* the ESP policy in transport mode to apply on that socket.
- *void setFlowLabel(int)*: set the flow label that is applied on this socket.
- *void setTrafficClass(int)*: set the traffic class that is applied on this socket.
- *void setNetAuthenticate(int)*: set value of the *IPV6_AUTH_NETWORK_LEVEL* option for this socket, *i.e.* the AH policy in tunnel mode to apply on that socket.
- *void setTransportAuthenticate(int)*: set value of the *IPV6_AUTH_TRANS_LEVEL* option for this socket, *i.e.* the AH policy in transport mode to apply on that socket.
- *void setNetEncrypt(int)*: set value of the *IPV6_ESP_NETWORK_LEVEL* option for this socket, *i.e.* the ESP policy in tunnel mode to apply on that socket.
- *void setTransportEncrypt(int)*: set value of the *IPV6_ESP_TRANS_LEVEL* option for this socket, *i.e.* the ESP policy in transport mode to apply on that socket.

SocketOptions6/SocketOptions

New Variables

- *IPV6_AUTH_TRANS_LEVEL*: set a particular AH policy in transport mode.
- *IPV6_AUTH_NETWORK_LEVEL*: set a particular AH policy in tunnel mode.

- *IPV6_ESP_TRANS_LEVEL*: set a particular ESP policy in transport mode.
- *IPV6_ESP_NETWORK_LEVEL*: set a particular ESP policy in tunnel mode.
- *IPV6_MULTICAST_IF* : set which outgoing interface on which to send multicast packets.

References

[1] A. M. Bakić, M. W. Mutka, and D. T. Rover. An on-line performance visualization technology. In *Heterogeneous Computing Workshop (HCW '99)*, pages 47–59, San Juan, Puerto Rico, April 1999. IEEE.

[2] D. Borman. TCP and UDP over IPv6 Jumbograms. RFC 2147, Network Working Group, May 1997.

[3] B. Carpenter, Y.-J. Chang, G. Fox, and X. Li. Java as a language for scientific parallel programming. *Lecture Notes in Computer Science*, 1366, 1998.

[4] C. Chassot, A. Loze, F. Garcia, L. Dairaine, and L. R. Cardenas. Specification and realization of the QoS required by a distributed interactive simulation application in a new generation internet. In M. Diaz, P. Owezarski, and P. Sénac, editors, *Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'99)*, volume 1718 of *Lecture Notes in Computer Science*, pages 75–91, Toulouse, France, October 1999.

[5] G. Chelius, G. Siu, and E. Fleury. Implementing IPv4 multicast with IPv6 sockets. proposition of changes to RFC 2292, 2000.

[6] G. Cizault. *IPv6: Théorie et pratique.* O'Reilly, 1998.

[7] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 2463, Network Working Group, December 1998.

[8] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Network Working Group, December 1998.

[9] Robert Fink. Network integration — boning up on IPv6 — the 6bone global test bed will become the new Internet. *Byte Magazine*, 23(3):96NA–3–96NA–8, March 1998.

[10] E. R. Harold. *Java Network Programming.* O'Reilly, 1997.

[11] R. Hinden, B. Carpenter, and L. Masinter. Format for Literal IPv6 Addresses in URL'. Internet Draft, 1999.

[12] R. Hinden and S. Deering. IP Version 6 addressing architecture. RFC 2373, Network Working Group, July 1998.

[13] C. Huitema. *IPv6: The New Internet Protocol.* Prentice Hall, 1996.

[14] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, Network Working Group, November 1998.

[15] David Powell. Group communication. *Communications of the ACM*, 39(4):50–97, April 1996. (special section Group Communication).

[16] M. J Skidmore, M. J. Sottile, and A. D. Cuny, J. E.and Malony. A prototype notebook-based environment for computational tools. In *High Performance Networking And Computing Conference*, Orlando, USA, November 1998.

[17] Richard W Stevens. *UNIX Network Programming.* Software Series. Prentice Hall PTR, 1990.

[18] W. Richard Stevens. *UNIX network programming: Networking APIs: sockets and XTI*, volume 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1998.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399